USB framework, IP core and related software

Salvador E. Tropea, Rodrigo A. Melo
Electrónica e Informática
Instituto Nacional de Tecnología Industrial
Buenos Aires, Argentina
Email: salvador@inti.gob.ar, rmelo@inti.gob.ar

***Abstract*—The *Universal Serial Bus* (USB) is currently the most common communication mechanism used for personal computer peripherals. USB replaced the traditional serial (RS-232) and parallel (IEEE 1284) ports.**

In this work we present a USB *core* that implements most of the features from the 2.0 specification, but not the isochronous mode. Additionally, we present the software tools developed to verify and use the *core*.

The *core* was verified in hardware using FPGAs and offers an ample variety of configurations.

I. Introduction

Most of today's personal computers (PCs) does not include serial and parallel communication ports. USB replaced them offering better performance, *plug and play* and other advantages.

Our team develop embedded systems that, in many cases, works as PC peripherals. Given the fact that USB is currently the most common communication mechanism we started the development of a USB *core*.

In this work we present the features of the developed *core*. Additionally, we present the auxiliary tools developed for verification purposes and to help in the use of the *core*. In section II we present the objectives of this project. The component parts of the *core* are depicted in section III. Section IV introduces the tools developed for verification purposes and the verification methodology. Implementation details and results are discussed in section V and finally we present the conclusions in section VI.

It is important to note that this document assumes the reader is already familiar with the USB terminology.

II. Objectives

We developed the *core* with the following goals:

- Sinthetizable for most of the FPGAs available in the market.
- Low cost, compared with an external solution.
- Allow for very compact configurations with minimal area and external components requirements.

To achieve a *core* synthetizable for a wide range of FPGAs, also opening the possibility to use it for ASICs, we used the standard VHDL 93 language.

To accomplish the second objective we took as reference the cost of a complete USB solution. The Cypress EX-USB FX2 line is a widely used USB solution, its cost is comparable to the cost of a 200,000 equivalent gates FPGA (*Spartan 3* 200, 3.840 LUTs+FFs). It imposed the first area constraint and also a limit to the cost of additional external parts needed to complement the FPGA.

The third objective was added to allow the use of the *core* as a tool to verify small peripherals implemented using low cost FPGAs. In our laboratory we have many *Spartan II* 100 [1] (2.400 LUTs+FFs) based development boards used for this purpose. It imposed a second area constraint.

III. Layers

To divide the problem into smaller parts that could be interchanged, and to allow for comparisons between our *core* and others, we adopted the following layers layout:

- **Electric layer**: voltage and/or current level conversion.
- **Physical layer or PHY**: we adopted the *USB 2.0 Transceiver Macrocell Interface* (UTMI) specification. This layer performs the parallel to serial conversion, encodes the bits to allow for clock synchronization and signals the start and end of packet. The reception module performs the reverse tasks.
- *Handshake* **layer or SIE**: usually known as *Serial Interface Engine* (SIE). It performs the basic USB protocol transactions.
- **SIE adaptation layer**: this layer adapts the SIE signals for the upper layer.
- **Protocol layer**: implements the higher part of the USB protocol, including the *plug and play*.
- **Application layer**: here we put the application, this is our device.

A. Electric layer

When we evaluate the transmitter point of view, the voltage levels specified by the USB standard for the lower speeds (low or LS 1,5 Mb/s and full or FS 12 Mb/s) are compatible with the levels provided by most FPGAs. On the contrary, when we do the same from the receptor point of view it is not true anymore. In laboratory conditions you can connect an FPGA to a USB *host* using a few resistors. On the other hand, when you must comply with the specification an external driver must be used.

For high speed (HS 480 Mb/s), USB uses currents for signaling, and hence an external driver is mandatory.

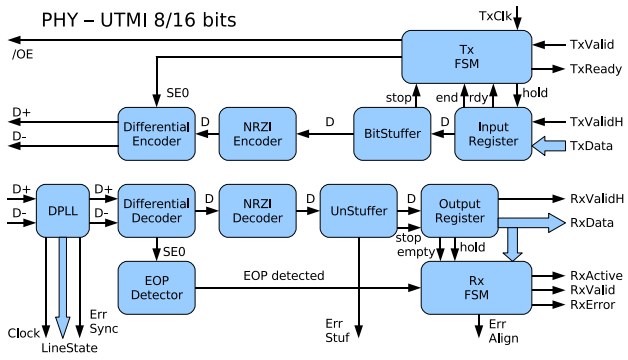

Fig. 1. PHY block diagram.

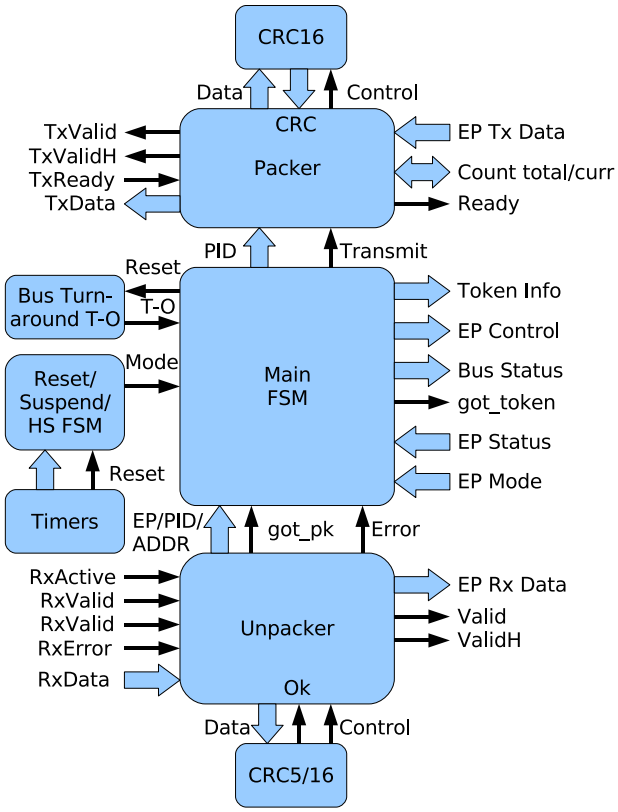

Fig. 2. SIE block diagram.

B. Physical layer or PHY

An UTMI compatible PHY was implemented, Fig. 1 shows a block diagram for our implementation. Its main features are: support for LS and FS, 8 and 16 bits interface and 48 MHz clock for FS (6 MHz for LS). We discarded an implementation for the HS mode using low cost FPGAs.

This layer performs two different tasks: packet reception and transmission. The receptor section takes care of clock recovery and synchronization (DPLL), bits decoding (NRZI and bitstuffing), serial to parallel conversion and start/end of packet detection. The transmitter section provides the parallel to serial conversion, start and end of packet signaling and bits encoding.

The receptor section detects bitstuffing, aligning and synchronization errors.

C. Handshake layer or SIE

This layer implements the basic USB protocol transactions. They comprise two or more packet exchanged between the *host* (master) and the *device* (slave). In our implementation we included the *control, interrupt* and *bulk* modes, but not the *isochronous* mode.

Our implementation was divided as shown in Fig. 2. The *packer* assembles the packets to be transmitted. It supports the DATA0 and DATA1 data packets and the ACK, NAK, STALL and NYET *handshake* packets. A 16 bits CRC (Cyclic Redundancy Code) is computed for the data packets. The *unpacker* disassembles the received packets. This module supports the IN, OUT and SETUP *tokens*; DATA0 and DATA1 data packets and ACK, NAK and STALL *handshake* packets. Other packet types are discarded. This block also verifies the CRC, 5 bits for *tokens* and 16 bits for data. Incomplete packet, CRC mismatch and wrong packet identifier errors are reported, not shown in the figure. The main state machine ensures that transactions are complete and selects the proper reply for each *token*. An independent state machine is used to detect the *reset* and *suspend bus* signaling. During the *reset*, and if the *core* was configured for HS, this state machine is in charge of the negotiation needed to switch from FS to HS. When the *host* needs to send data (SETUP and OUT *tokens*), or when the SIE is waiting for an ACK, the USB specification defines a time-out. The *Bus Turn-around T-O* block is used to detect this time-out. The *Timers* block is used to generate various timing signals: 2,5 μs (*reset*), 100 μs and 1 ms (negotiation) and 3 ms (*suspend* and negotiation).

The *Token Info bus* includes information about the received *token* (*endpoint* number, packet type, etc.), *EP Control* includes signals to enter/exit to/from the STALL mode, *Bus Status* informs about the *bus* state (*idle, reset, active*), *EP Status* is used to indicate the current *endpoint* status (available, STALL, not yet available) and the data packet type (DATA0/1) that we expect to receive or want to transmit, finally, *EP Mode* indicates which modes are supported by the currently selected *endpoint* (IN, OUT, SETUP).

Data *buses* are configurable for 8 and 16 bits. Additionally, the clock for this module can be configured to be the same or just half of the clock used for the PHY.

D. SIE adaptation layer

Most of the available USB *cores* [2] [3] [4] [5] [6] include control and status registers, along with the buffers used for the received data and data to be transmitted, in this layer. This is a very flexible approach when the *core* is used by a microcontroller.

We wanted to optimize the use of resources, and determined that many cases can be solved without the need of a full CPU. We also found that in many cases the *buffers* are not needed, usually when dealing with small or volatile data. For

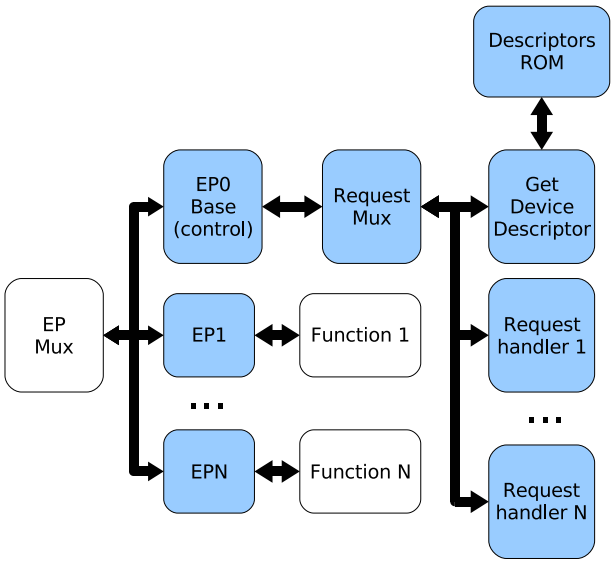

Fig. 3. Modular architecture for *enpoints* and *requests*.

these reasons we simplified this layer to only include the multiplexers used to route the data for the *enpoints*.

E. Protocol layer

Among the most important USB features are the *plug and play* support, multiple configurations for the same device and support for more than one functionality. To achieve these features the *host* must be able to consult the *device*. The *host* can use *requests* that are replied by the *device* with special structures, called *descriptors*, containing the needed information.

The interpretation of the *host requests* and *descriptors* handling can be implemented in software [4] [2]. This method is very versatile because we do not need to modify the *core* for each device, only the software. In other cases a fixed number of *enpoints* are provided and this layer is joined with the previous [3] [6]. This approach simplifies the *software*, but requires changes in the *core* if the provided *enpoints* does not match the needs of the device that we are developing.

To achieve small USB implementations we fully implemented this layer in hardware. The supported *requests* are device dependent, and we also wanted to avoid limitations in the number and type of used *enpoints*. For these reasons we designed a modular architecture. This approach supports the addition of modules that implements different *request* and *enpoints*.

To simplify the developing task we identified the most commonly used *requests* and encapsulated them in a *core* (*EP0 Base*). The user only needs to add the missing *requests*. Fig. 3 shows a simplified block diagram for the designed modular architecture.

F. Application layer

This is where the functionality for our device is implemented. It can be implemented in a very compact way thanks to the hardware implementation of the previous layer. In many cases this layer only adds a few registers to the hardware that implement our device. When an *endpoint* needs data buffering the buffers can be implemented in this layer, but in any other case we avoid the implementation of unneeded buffers.

IV. Validation and complementary tools

A. Validation tools

USB is complex, and, as mentioned above, composed by various layers. We developed automatic testbenches to verify each part of the *core* separately and then working in conjunction. To achieve a modular testing approach we used the UNIX strategy: many small tools that can solve simple tasks, but can also work together to solve bigger problems.

We developed the tools using C and C++ languages and *pipes* for communication between them.

To validate the PHY we developed the following tools:

- **gen-dpll-diff**: generates a random bit stream, using a clock with a drift. The maximum deviation for the clock was adjusted according to the specification. Its output is differential and was used to verify the *DPLL* and the *Differential Decoder*.
- **nrzi_en**: encodes the bits in NRZI (*Non return to zero, inverted*) format. It is used to verify the *NRZI Encoder*.
- **nrzi_de**: decodes the NRZI encoded bits. It is used to verify the *NRZI Decoder*.
- **bitstuffer**: adds the bits indicated by the specification. It is used to verify the *BitStuffer*.
- **bitunstuffer**: removes the bits added by the bitstuffer. It is used to verify the *UnStuffer*.
- **usb-make-packet**: assembles and disassembles USB packets. It adds or removes the start of packet (SYNC) and the end of packet (EOP). To properly encode or decode the bits this tool uses nrzi_en, nrzi_de, bitstuffer and bitunstuffer. It is used to verify the complete PHY.

For the SIE validation:

- **crc**: computes the 5 and 16 bits USB CRC.
- **usb_to_bits**: is used to generate USB transactions. It interprets an input text where we describe the wanted transactions, and optionally the errors to be introduced. It uses usb-make-packet and crc to generate the USB packets, and is used to verify the *Unpacker* and the whole SIE.

B. Tools for validation and development

We developed two more tools that not only helped during the verification but also during the development of USB devices:

- **hid-report**: HID (Human Interface Device) devices can inform to the *host* the used data format. HID examples are keyboards, mouses and joysticks. The HID specification uses a pseudo-language for the examples to show how

this information is encoded. This program can interpret the pseudo-language and compile it to the equivalent *descriptor*.

- **usb-descriptor**: *descriptors* are used to inform various details to the *host*. When developing USB devices the manual creation of such *descriptors* becomes a complex, and error-prone, task. The problem is even worst when we need to modify some detail, like adding an *endpoint*. In this case we could forget to modify all the affected bytes. To simplify this task usb-descriptor can interpret a high level description of our device and the create the proper *descriptors*. For the HID *descriptors* usb-desciptor can make use of hid-report. Additionally, this program generates the VHDL code for the *descriptors* ROM and a *package* containing the memory addresses for each *descriptor*.

Using usb-descriptor and usb_to_bits we verified complete USB devices injecting real world transactions. To create a complete real world initialization we monitored a USB keyboard using a digital oscilloscope. We used tools like usb-make-packet to disassemble the captured packets. Finally, we wrote the complete initialization sequence using the usb-to-bits and usb-descriptor languages. Using this information we created realistic transactions for our devices. We also altered these sequences to verify the behavior of the devices when the packets contained various kinds of errors.

V. Implementation and results

A. Implementation

Our *core* was implemented using standard VHDL 93, and avoiding the use of vendor or device specific details. We used GHDL [7] 0.27 and other tools recommended by the FPGALibre [8] project for simulation and validation. The hardware validation was performed using Xilinx's *Spartan II* and *Spartan 3* FPGAs and the ISE WebPack 9.2.03i J.39 software.

For LS and FS an ISP1106 driver connected to a *Spartan II* was used, its cost is approximately 5% of a complete solution. We verified that under laboratory conditions, and using a 1.5 m wire, the FPGA can be connected directly to the *host* computer. For HS a Cypress CY7C68000 UTMI connected to a *Spartan 3*, using a 16 bits bidirectional data bus at 30 MHz, was used. This component costs approximately 15% of a complete solution. In both cases the printed circuit board area consumed by external parts was less than the area needed for a complete external solution. The HS solution could be implemented using a *Spartan II* device, but we have the external UTMI in a development board for *Spartan 3*.

As *host* we used a personal computer running the Debian [9] GNU [10] /Linux operating system. To control our devices from the *host* we created programs using C++ and two different Linux APIs: hiddev and libusb.

B. Results

We developed the following demonstration devices:

- **Analog joystick to USB adaptor**: it allows the use of an old analog joystick with a modern computer. This device implements the HID and have only one *endpoint*. To convert the analog signals we used two external comparators and capacitors. The implemented A/D converter measures the time needed to charge the capacitor through the potenciomenter of the joystick. This device consumed 708 LUTs and 427 FFs (493 *slices*) along with one BRAM. It was verified using GNU/Linux and Windows XP.
- **Generic HID**: to experiment with the Linux HID API we developed a *core* to control the development board LEDs and also monitor the board switches and buttons. It has two *enpoints* and consumed 692 LUTs and 402 FFs (488 *slices*) plus one BRAM.
- **Generic USB device**: to experiment with the Linux libusb API we developed a similar *core*, but without implementing the HID details. Additionally, we included a third *endpoint* using the *bulk* mode to read the *descriptors* ROM. Three versions of the *core* were implemented, two using FS and one HS. The FS version was implemented using 8 and 16 bits data buses. We obtained the following results: FS/8 bits 584 LUTs and 349 FFs (418 *slices*), FS/16 bits 925 LUTs and 466 FFs (591 *slices*) and HS 860 LUTs y 347 FFs (511 *slices*).
- **USB to WISHBONE bridge**: we use the WISHBONE *bus* for most of our peripherals and hence this bridge was an interesting choice. Using it a PC can control the WISHBONE *bus* using the USB. It allows the control of any WISHBONE peripheral from the PC and is very useful for verification purposes. For demonstration purposes we connected a small peripheral to the WISHBONE *bus*. This small device can be used to control the development board LEDs and also monitor the board switches and buttons. We implemented two versions of the *core*: FS 716 LUTs and 404 FFs (496 *slices*) and HS 927 LUTs and 396 FFs (571 *slices*).

C. Other implementations

To fairly compare the above mentioned results we should implement the same applications using other cores. This is time consuming and expensive. We could compare blocks with the same functionality instead, but this is a partial solution and not all implementations use the same layers or have the same functionality. For these reasons we only present an approximated comparison.

OpenCores' usbhostslave [5] *core* includes a mouse implementation. This application can be compared with our *joystick*. The usbhostslave synthesis resulted in 2511 LUTs and 1631 FFs (1704 *slices*) for a *Spartan 3*. These values were corrected to discount 512 LUTs used as dual ported memory, we estimate this memory could be mapped to one BRAM. This *core* includes a PHY, a SIE, interface registers and FIFOs and the mouse device.

In other cases we did not have full devices and hence we measured the area for the SIE and the adaptation layer.

Note that this value does not include the PHY, protocol layer and the device: Xilinx LogiCORE [2] 2340 LUTs and 610 FFs (1235 *slices*) 4 BRAMs (8 *enpoints* using *Spartan 3* or better, OPB *bus* interface), OpenCores usb [4] 2899 LUTs and 1712 FFs (1985 *slices*) using external memory (16 *enpoints* using *Spartan 3*, WISHBONE *bus* interface) and Aeroflex/Gaisler GRUSBDC [11] 3500 LUTs and 2 BRAMs using the minimum configuration (no FFs information, up to 32 *enpoints* using *Spartan 3*, Amba *bus* interface). Another example is the ComBlock [3] *core*. It implements the protocol layer in *hardware* using a very rigid approach and consumes 1449 LUTs and 620 FFs (854 *slices*) 5 BRAMs (4 *enpoints* using Virtex-2). All the above mentioned *cores* have support for HS using an external UTMI chip.

Another *core* that could be used as reference is the OpenCores usb1_fun [6]. It includes a PHY, the SIE and the protocol layer implemented in *hardware*. Its protocol layer supports 6 fixed *enpoints*. No application example is provided and the changes needed in order to implement an HID device are very important. This *core* does not have HS support and consumes 978 LUTs and 518 FFs (670 *slices*) and one BRAM.

VI. Conclusions

We obtained a compact USB implementation that enables the use of small FPGAs like *Spartan II*. A full USB application consumed only 35% of a *Spartan II* 100.

The number and cost of external components was reduced to a minimum thanks to the implementation of a PHY with LS and FS support. The external support driver needed is cheap and has a small footprint, i.e. ISP1106. Under laboratory conditions we verified that an FPGA can be directly connected to the USB.

The cost of the consumed area is small, 22% of a *Spartan 3* 200, and consequently provides a better solution, when compared with a full external solution. The use of the UTMI standard enabled the use of an HS external PHY for applications demanding high throughput.

The hid-report and usb-descriptor tools considerably reduced the development time for the demonstration applications. Additionally, the use of small and modular tools enabled a good code reuse. The development tools and methodologies proposed by the FPGALibre project proved to be suitable for this project.

Even when we can not provide a fair comparison with other implementations, we can conclude that our *core* allows the creation of devices with a smaller area consumption. Only one third of the area is needed if we compare the above mentioned mouse [5] with our *joystick*. Even when comparing only a portion of an application, all the studied *cores* consume more area. After an analysis of the architectures we conclude that the main difference is in the layers between the SIE and the application. Our choice not only saves an important amount of resources, but also simplifies the application layer.

References

[1] D. J. Brengi, S. E. Tropea, and J. P. D. Borgna, "Tarjeta de diseño abierto para desarrollo y educación," in *2007 3rd Southern Conference on Programmable Logic Designer Forum Proceedings*, Mar del Plata, 2007, pp. 57–60.

[2] (2008, Dec.) OPB universal serial bus device (v1.00a). Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/usb_ds591.pdf

[3] (2008, Dec.) USB 2.0 interface user manual. ComBlock. [Online]. Available: http://www.comblock.com/ download/USB20_UserManual.pdf

[4] R. Usselmann. (2008, Dec.) USB 2.0 function core. OpenCores.org. [Online]. Available: http://www.opencores.org/projects.cgi/web/usb

[5] S. Fielding. (2008, Dec.) USB 1.1 host and function ip core. OpenCores.org. [Online]. Available: http://www.opencores.org/projects.cgi/web/ usbhostslave

[6] R. Usselmann. (2008, Dec.) USB 1.1 function ip core. OpenCores.org. [Online]. Available: http://www.opencores.org/projects.cgi/web/usb1_funct/overview

[7] T. Gingold. (2008, Dec.) A complete VHDL simulator. [Online]. Available: http://ghdl.free.fr/

[8] S. E. Tropea, D. J. Brengi, and J. P. D. Borgna, "FPGAlibre: Herramientas de software libre para diseño con FPGAs," in *FPGA Based Systems*. Mar del Plata: Surlabs Project, II SPL, 2006, pp. 173–180.

[9] I. Murdock *et al.* Debian gnu/linux operating system. [Online]. Available: http://www.debian.org/

[10] R. M. Stallman *et al.* The GNU project. [Online]. Available: http://www.gnu.org/

[11] (2008, Dec.) USB 2.0 device controller. Aeroflex / Gaisler. [Online]. Available: http://www.gaisler.com/